CS 540 Introduction to Artificial Intelligence
Game II

Yingyu Liang
University of Wisconsin-Madison
Nov 30, 2021

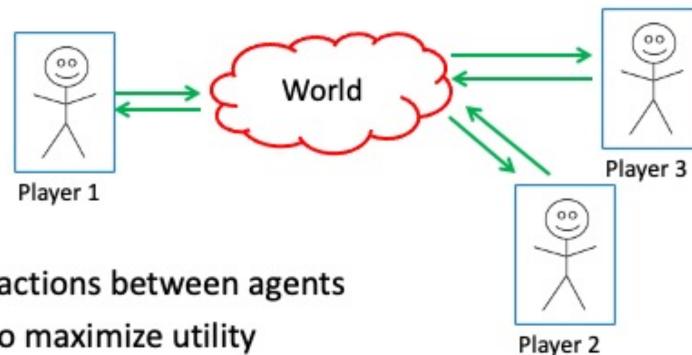
Based on slides by Fred Sala

Outline

- Review of game theory basics
  - Properties, sequential games
- Speeding up sequential game search
  - Heuristics, pruning, random search
- Simultaneous Games
  - Normal form, strategies, dominance, Nash equilibrium

Review of Games: Multiple Agents

Games setup: **multiple** agents



- Now: interactions between agents
- Still want to maximize utility
- **Strategic** decision making.

The general model of games:

1. We have multiple agents/players that interact with each other and with the world
2. Each player has its own reward function, and each player just wants to maximize its own reward.
3. However, the reward of any single player depends on the actions of all players and the state of the world. So the reward function captures the interactions between the players and the world.
4. Because the player wants to maximize its reward, we say the player is strategic or rational or selfish.

Review of Games: Properties

Let's work through **properties** of games

- **Number** of agents/players
- State & action spaces: **discrete** or **continuous**
- **Finite** or **infinite**
- **Deterministic** or **random**
- **Sum**: zero or positive or negative
- **Sequential** or **simultaneous**

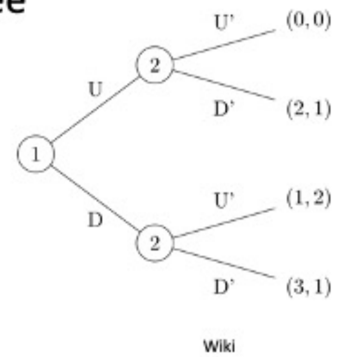
Wiki

Games have different properties. We can divide the games into different categories based on these properties, and then study each category.

Sequential Games

Games with multiple moves

- Represent with a **tree**
- Find strategies: perform search over the tree



Wiki

II-Nim: Example Sequential Game

2 piles of sticks, each with 2 sticks.

- Each player takes one or more sticks from pile
- Take last stick: lose

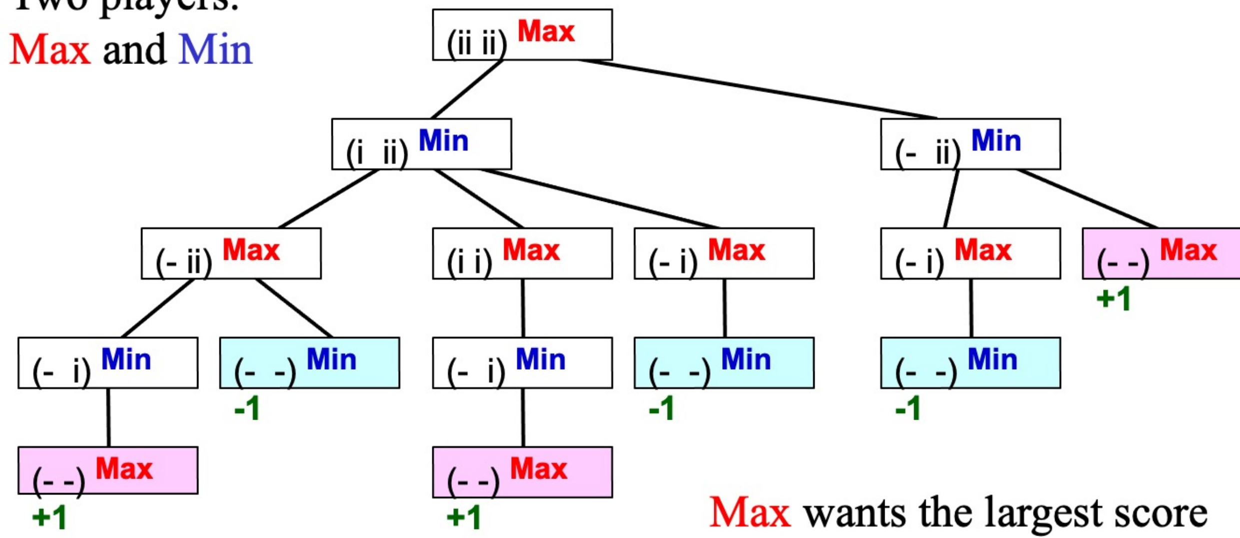
(ii, ii)

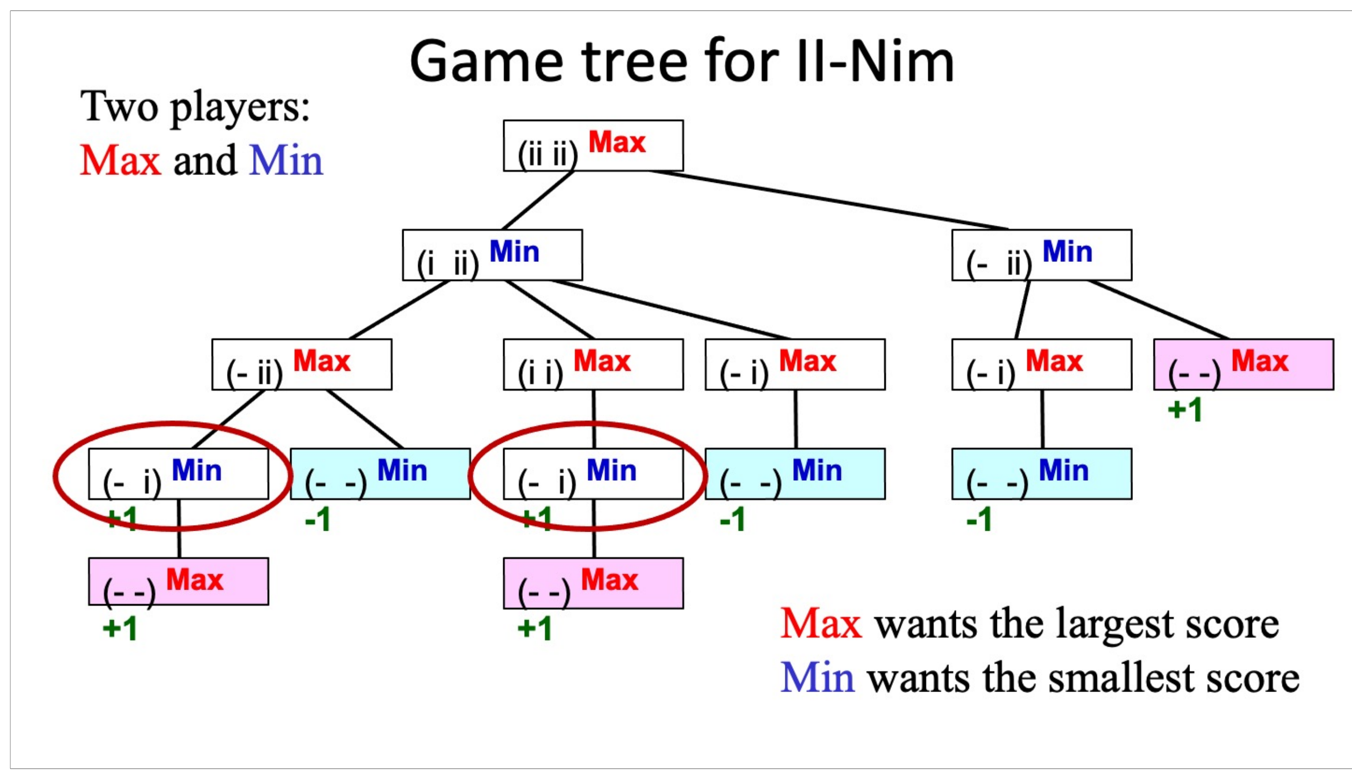
- Two players: Max and Min
- If Max wins, the score is +1; otherwise -1
- Min's score is –Max's
- Use Max's as the score of the game

There are two players. They take turns to make a move. The first player makes the first move, and then the second player makes a move, and then the first player, and so on.

Since the two players' scores sum up to 0, when we describe the outcome of the game, we can just mention one player's score. The other player's score is just the negation. The convention is to use the first player's score. We just define the score of the game to be that of the first player.

So the first player wants to maximize the score of the game: that's why we call the first player Max. The second player wants to maximize its own score, that is to minimize the score of the game, so we call the second player Min.


Game tree for II-Nim
Two players:
Max and Min
(ii ii) Max
(i ii) Min
(- ii) Min
(- ii) Max
(i i) Max
(- i) Max
(- i) Max
(- -) Max
+1
(- i) Min
(- -) Min
-1
(- i) Min
(- -) Min
-1
(- -) Min
-1
(- -) Max
+1
(- -) Max
+1
Max wants the largest score
Min wants the smallest score



Recall the game-theoretical value of a current state: assume from this state the two players will play optimally, and at the end we will reach a terminal state. The score of the terminal state is defined to be the game-theoretical value of the current state.

We can compute the value from bottom up.

Terminal state: = the score of the game in this terminal state
A state where Min is going to play: take the minimum value of the children
A state where Max is going to play: take the maximum value of the children

Game tree for II-Nim

Two players:
Max and Min

(ii ii) Max

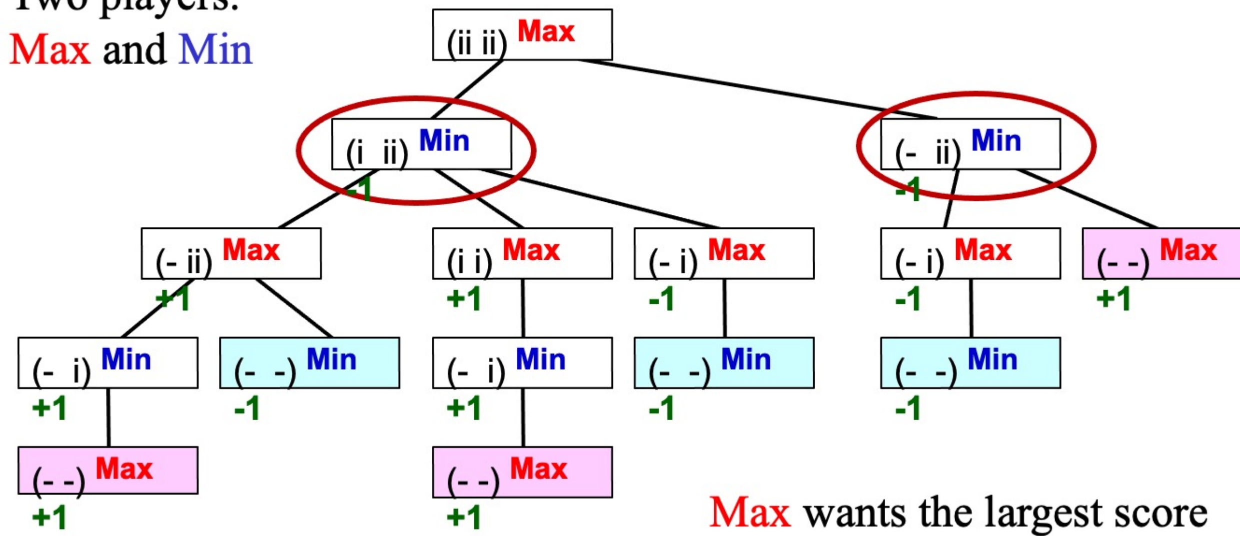
(i ii) Min
(- ii) Min

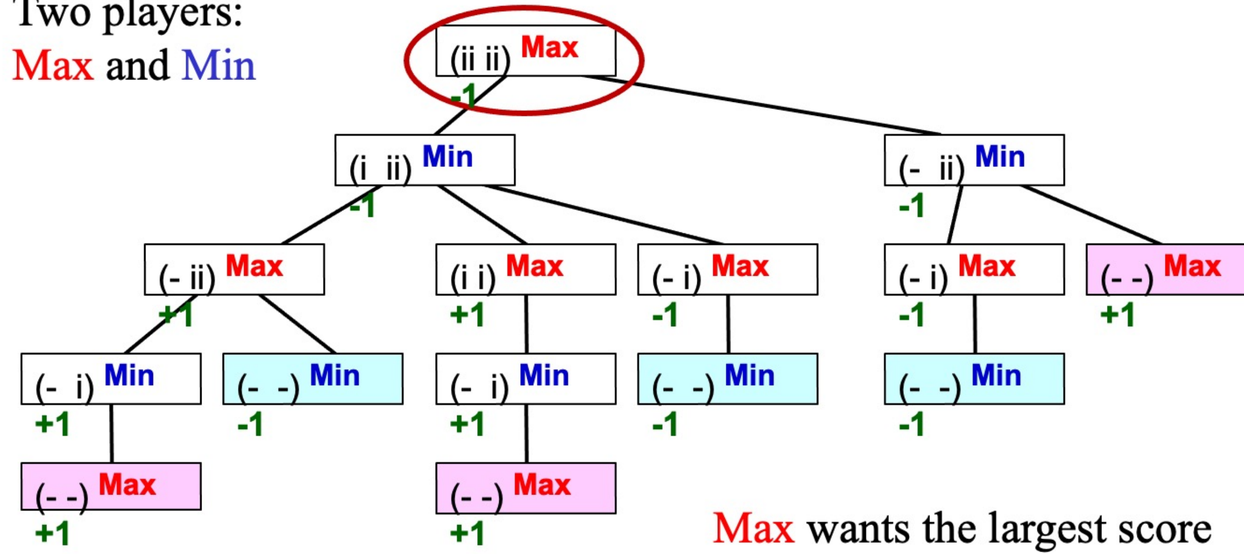
(- ii) Max +1
(i i) Max +1
(- i) Max -1
(- i) Max -1
(- -) Max +1

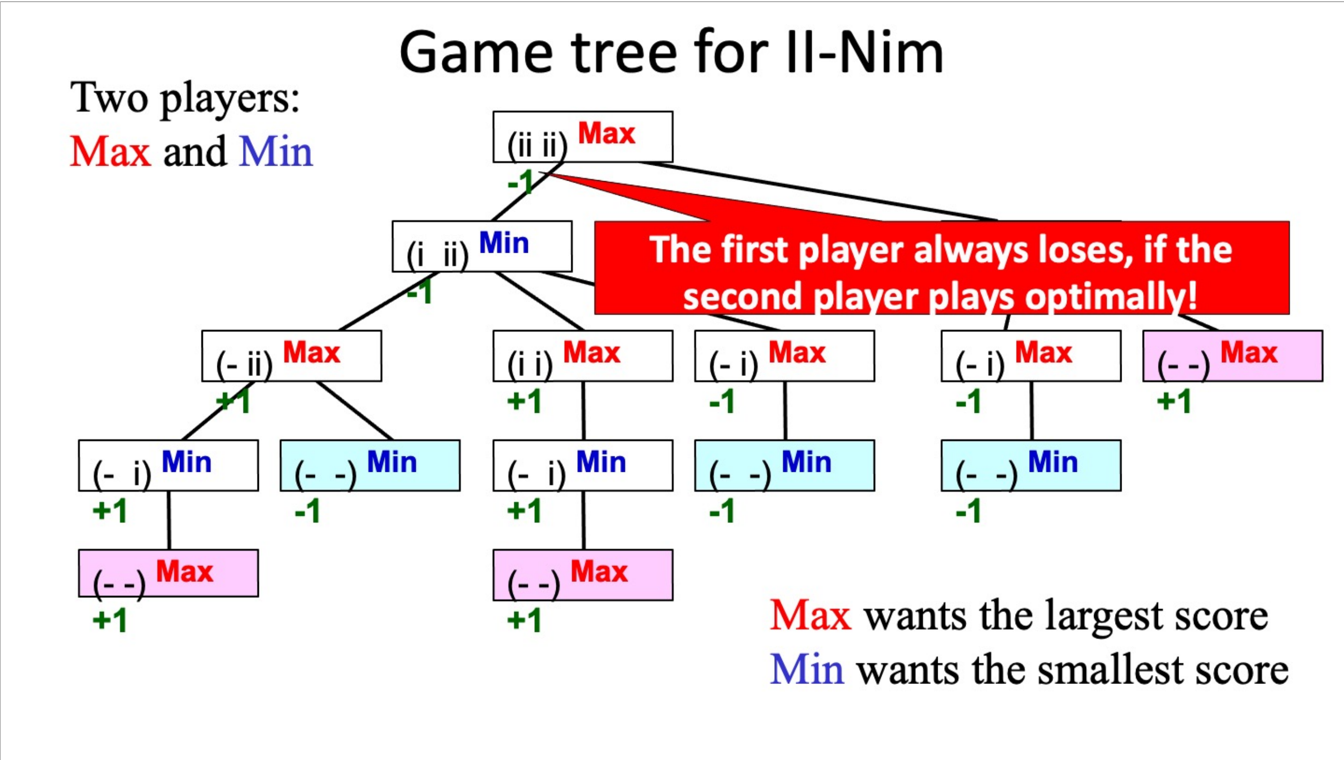
(- i) Min +1
(- -) Min -1
(- i) Min +1
(- -) Min -1
(- -) Min -1

(- -) Max +1
(- -) Max +1

Max wants the largest score
Min wants the smallest score


Game tree for II-Nim
Two players:
Max and Min
(ii ii) Max
(i ii) Min
-1
(- ii) Min
-1
(- ii) Max
+1
(i i) Max
+1
(- i) Max
-1
(- i) Max
-1
(- -) Max
+1
(- i) Min
+1
(- -) Min
-1
(- i) Min
+1
(- -) Min
-1
(- -) Min
-1
(- -) Max
+1
(- -) Max
+1
Max wants the largest score
Min wants the smallest score


Game tree for II-Nim
Two players:
Max and Min
(ii ii) Max
-1
(i ii) Min
-1
(- ii) Min
-1
(- ii) Max
+1
(i i) Max
+1
(- i) Max
-1
(- i) Max
-1
(- -) Max
+1
(- i) Min
+1
(- -) Min
-1
(- i) Min
+1
(- -) Min
-1
(- -) Min
-1
(- -) Max
+1
(- -) Max
+1
Max wants the largest score
Min wants the smallest score



Interesting conclusion: the first player will always lose if the second player plays optimally. From the rule of the game we don't immediately see who will win. However, with the game tree and the game-theoretical value, we can obtain the highly non-trivial conclusion that the first player will always lose if the second player plays optimally.

Two other important implications:
We can compute the game-theoretical values easily from bottom up.
Once we have the values of the children of a current state, then we know which is the best action. That is, to play the game, all we need is to compute the values of the current state.

Minimax Algorithm

```
function Max-Value(s)
inputs:
      s: current state in game, Max about to play
output: best-score (for Max) available from s

      if ( s is a terminal state )
      then return ( terminal value of s )
      else
                  α := – infinity
                  for each s’ in Succ(s)
                     α := max( α , Min-value(s’))
      return α
```

```
function Min-Value(s)
output: best-score (for Min) available from s

      if ( s is a terminal state )
      then return ( terminal value of s)
      else
                  β := infinity
                  for each s’ in Succs(s)
                     β := min( β , Max-value(s’))
      return β
```

Time complexity?

- $O(b^m)$

Space complexity?

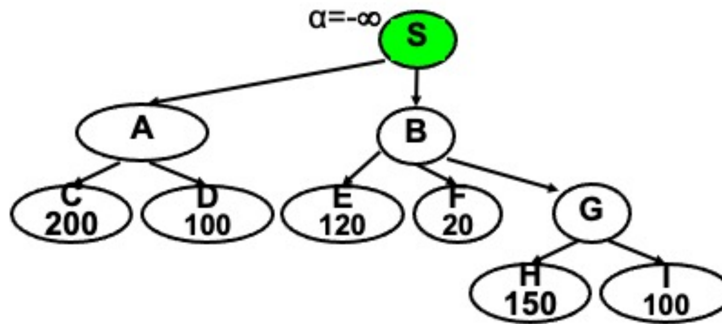
- $O(bm)$

The minimax algorithm replaces the bottom-up computation with recursion.

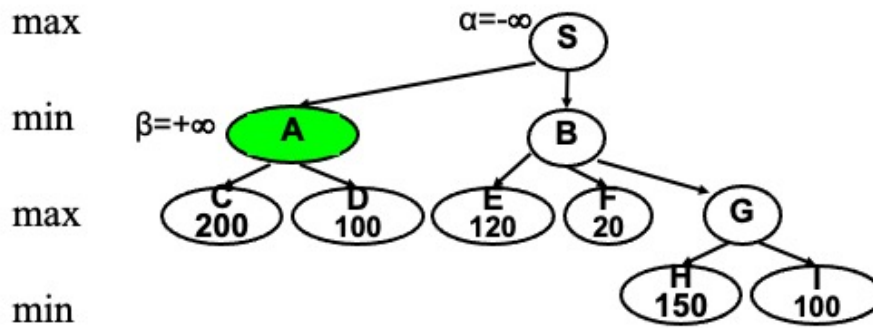
The key idea of recursion: we assume that smaller problems are already solved, and we want to use the solutions for the smaller problems to solve the current problem. Here, the smaller problems are the values of the children, and the current problem is the value of the current state.

On a state where Max is going to play:
it’s terminal then we can return the terminal score which is the value by definition
If not terminal, just take the maximum of the values of the children (here we pretend that we have already solve the smaller problems of computing the values of the children)
This is the Max-Value function. Similar for the Min-Value function that computes the value of a state where Min is going to play.

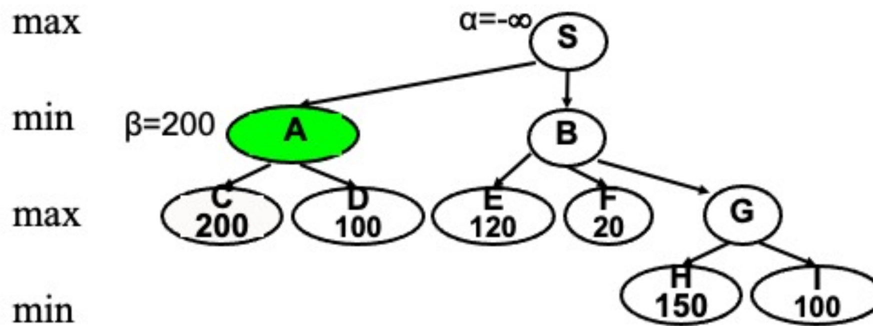
Minimax algorithm in execution



Minimax algorithm in execution



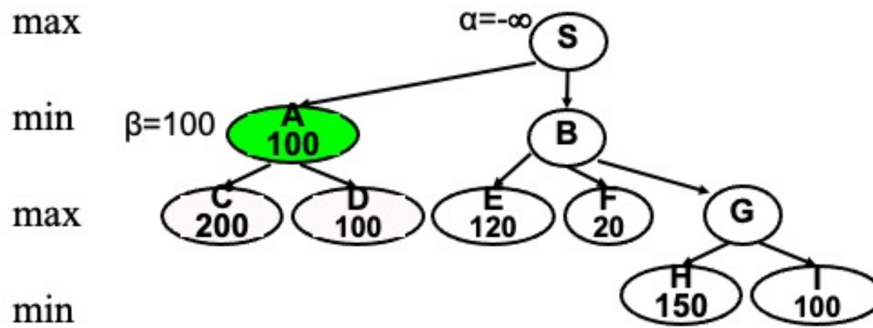
Minimax algorithm in execution



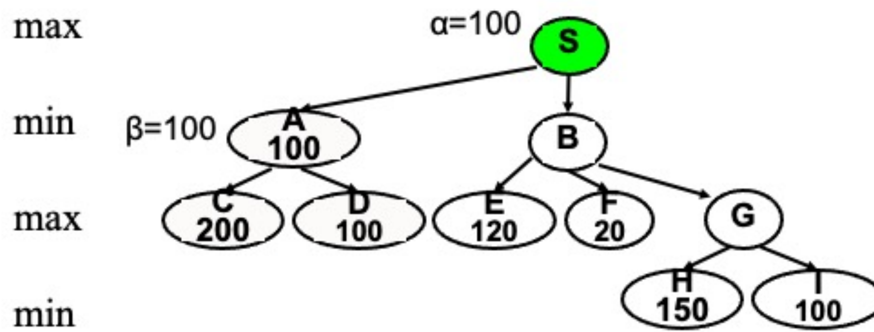
The execution on the
terminal nodes is omitted.

Beta is recording the minimum value of the children we obtain up to the current time point.

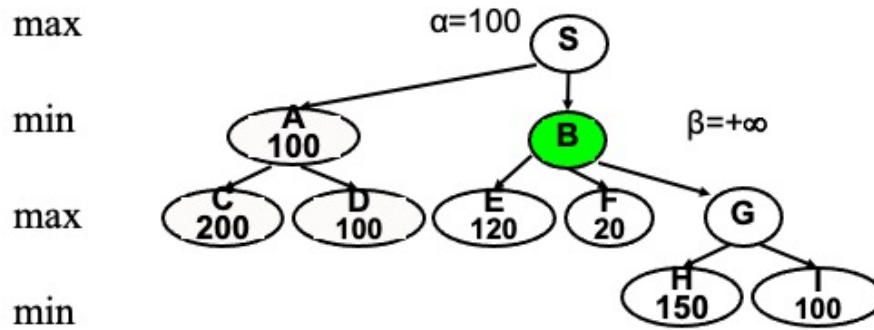
Minimax algorithm in execution



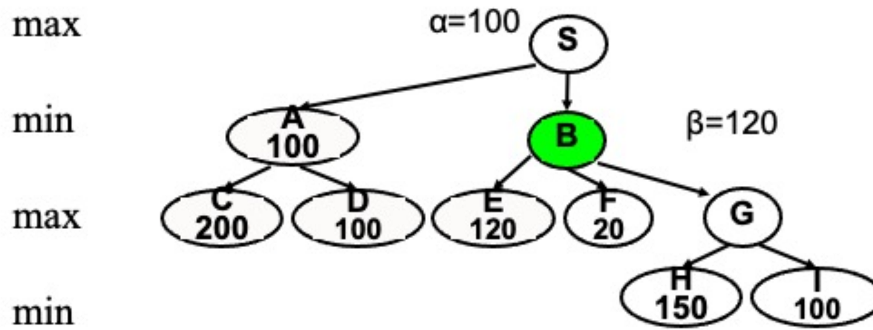
Minimax algorithm in execution



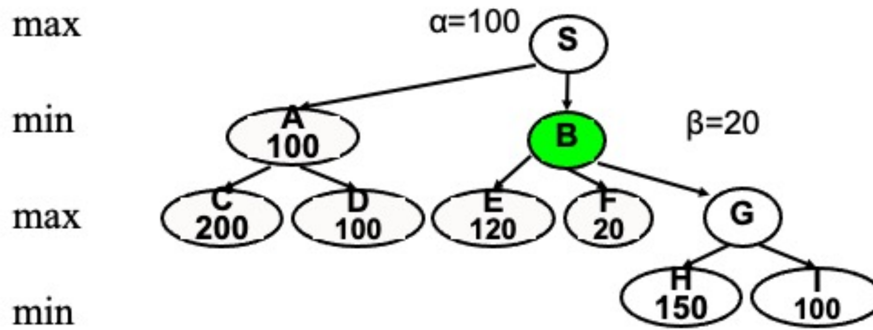
Minimax algorithm in execution



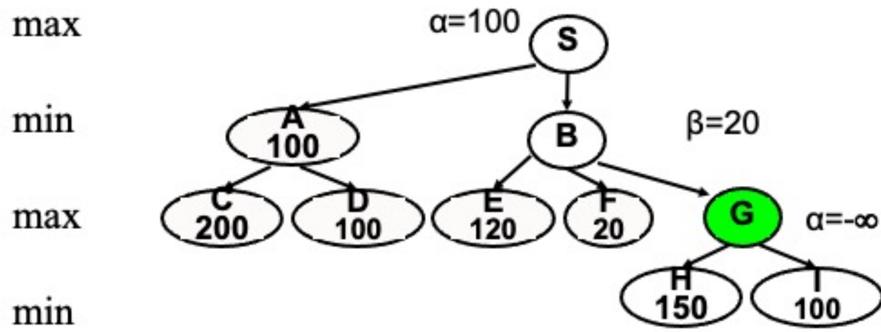
Minimax algorithm in execution



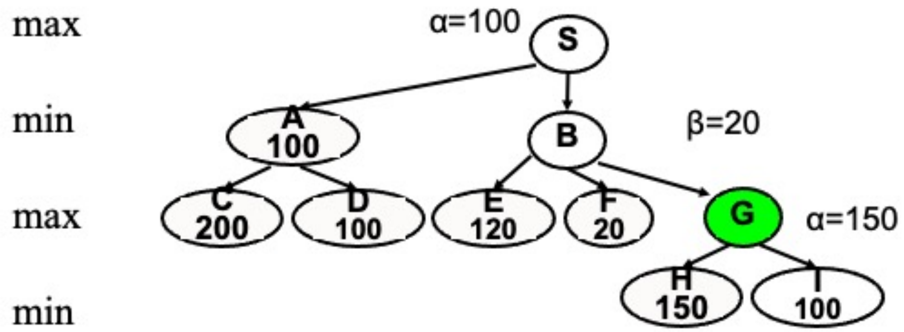
Minimax algorithm in execution



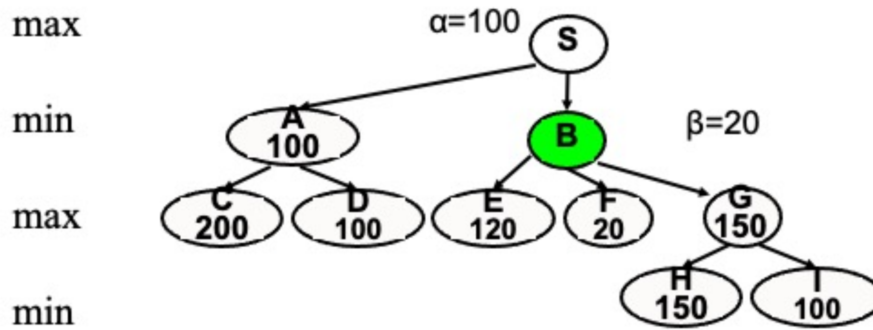
Minimax algorithm in execution



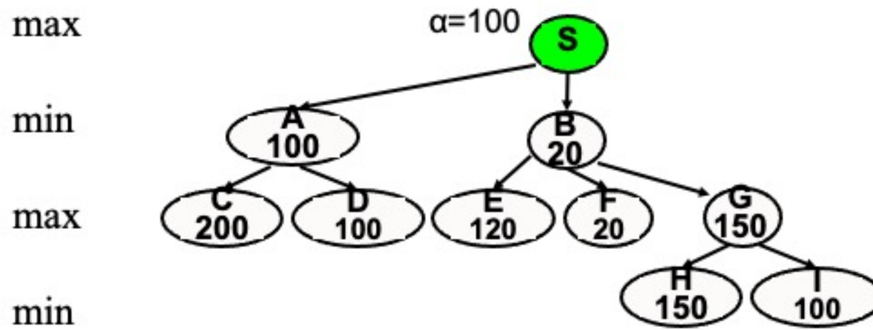
Minimax algorithm in execution



Minimax algorithm in execution



Minimax algorithm in execution



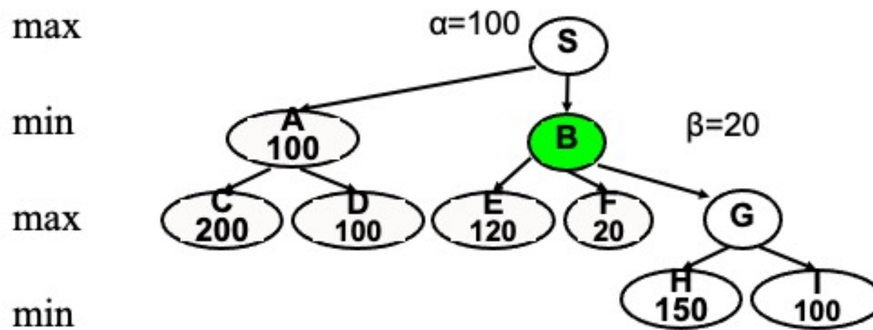
Can We Do Better?

One **downside**: we had to examine the entire tree

An idea to speed things up: **pruning**

- Goal: want the same minimax value, but faster
- We can get rid of bad branches: when we are sure that pruning them doesn't affect the minimax value

Minimax algorithm in execution



Example: when beta on B node is updated to 20. Then we know that G can be pruned.

1. No matter what G's value is, B's value is <= 20
2. Then because beta < alpha, alpha will not be updated.
3. So we don't need to compute B's exact value anymore; no need to check the remaining chil

Alpha-beta pruning

```
function Max-Value (s,α,β)
inputs:
     s: current state in game, Max about to play
     α: best score (highest) for Max along path to s
     β: best score (lowest) for Min along path to s
output: min(β , best-score (for Max) available from s)
     if ( s is a terminal state )
     then return ( terminal value of s )
     else for each s' in Succ(s)
       α := max( α , Min-value(s',α,β))
       if ( α ≥ β ) then return β   /* alpha pruning */
     return α
```

```
function Min-Value(s,α,β)
output: max(α , best-score (for Min) available from s )
     if ( s is a terminal state )
     then return ( terminal value of s)
     else for each s' in Succs(s)
       β := min( β , Max-value(s',α,β))
       if (α ≥ β ) then return α   /* beta pruning */
     return β
```

Starting from the root:
Max-Value(root, $-\infty$, $+\infty$)

Meaning of alpha beta: up to this point, the best Max/min can force; alpha is lower bound of the game value, beta is upper bound. Begin with infinity.

The pseudo code of Min-Value: similar to minimax algo, but can stop when alpha>= beta.

Why one can do the pruning: e.g., α := max(α , Min-value), so we can stop Min-value if we know it's smaller than α. Refer to the example in the previous slide.

Alpha-Beta Pruning

How effective is **alpha-beta pruning**?

- Depends on the order of successors!
  - Best case, the #of nodes to search is O($b^{m/2}$)
  - Happens when each player's best move is the leftmost child.
  - The worst case is no pruning at all.

- In DeepBlue, the average branching factor was about 6 with alpha-beta instead of 35-40 without.

Minimax With Heuristics

Note that long games are yield huge computation

- To deal with this: limit ***d*** for the search depth
- **Q**: What to do at depth *d*, but no termination yet?
  - **A**: Use a heuristic evaluation function *e(x)*

```
function MINIMAX(x, d) returns an estimate of x's utility value
  inputs: x, current state in game
          d, an upper bound on the search depth
  if x is a terminal state then return Max's payoff at x
  else if d = 0 then return e(x)
  else if it is Max's move at x then
      return max{MINIMAX(y, d−1) : y is a child of x}
  else return min{MINIMAX(y, d−1) : y is a child of x}
```

Credit: Dana Nau

The time complexity of the minimax algo is not good: exponential in m, the number of steps.

We can address this by limiting the search depth, similar to what we have done in iterative deepening. That is, if we want to compute the value of a current state, we only go down the current state for depth d.

The question is: what if we get to a node at depth d but it’s not a terminal state? What value should we return? We can just use some estimation.

Heuristic Evaluation Functions

- e(x) often a weighted sum of features (like our linear models)

$$e(x) = w_1 f_1(x) + w_2 f_2(x) + \ldots + w_n f_n(x)$$

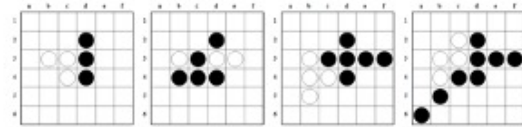
- Chess example: $f_i(x)$ = **difference** between number of white and black, with *i* ranging over piece types.
  - Set weights according to piece importance
  - E.g., 1(# white pawns - # black pawns) + 3(#white knights - # black knights)

A common way to design the heuristic function: linear model, which is a weighted sum of some designed features.

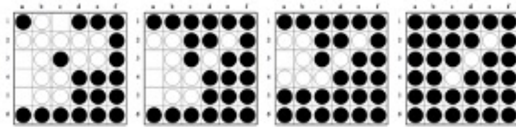
The features are typically some intuitive important information about the state, like the difference of white and black pieces in Chess.
The weights are set according to the importance of the features.

Going Further

- Monte Carlo tree search (MCTS)
  - Uses random sampling of the search space
  - Choose some children (heuristics to figure out #)
  - Record results, use for future play
  - Self-play
- AlphaGo and other big results!



The agent (Black) learns to capture walls and corners in the early game



The agent (Black) learns to force passes in the late game

Credit: Surag Nair

Break & Quiz

Q 1.1: During minimax tree search, must we examine every node?

- A. Always
- B. Sometimes
- C. Never

Break & Quiz

Q 1.1: During minimax tree search, must we examine every node?

- A. Always
- **B. Sometimes**
- C. Never

Break & Quiz

Q 1.1: During minimax tree search, must we examine every node?

- A. Always (No: consider alpha-beta pruning: consider layer *k*, where we take the max of all the mins of its children at layer *k+1*. If the current value of a min node at *k+1* already smaller than the current max, we don't need to continue the minimization.)
- **B. Sometimes**
- C. Never (No: the event above may simply not happen).

Another Example: Prisoner's Dilemma

Famous example from the '50s.

Two prisoners A & B. Can choose to betray the other or not.

- A and B both betray, each of them serves two years in prison
- One betrays, the other doesn't: betrayer free, other three years
- Both do not betray: one year each

Properties: **2-player, discrete, finite, deterministic, negative-sum, simultaneous**

Simultaneous Games

The players make moves simultaneously

- Can express reward with a simple diagram
- Ex: for prisoner's dilemma

Player 2 Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

Normal Form

Mathematical description of simult. games. Has:

- n players $\{1,2,\ldots,n\}$
- Player i strategy a_i from A_i. **All**: $a = (a_1, a_2, \ldots, a_n)$
- Player i gets rewards $u_i(a)$ for any outcome
  - **Note**: reward depends on other players!
- Setting: all of these spaces, rewards are **known**

In general, we can have a math description of simultaneous games called normal form.

Suppose we have n players, and player I can choose action/strategy from a set A_i. Let a denote the set of actions by all players. Then let u_i(a) denote the reward/utility for player i when the actions are a.

We can present u_i's as a n-dimensional array, as in the Prisoners' dilemma.

Example of Normal Form

Ex: Prisoner's Dilemma

Player 2 / Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

- 2 players, 2 actions: yields 2x2 matrix
- Strategies: {Stay silent, betray} (i.e, binary)
- Rewards: {0,-1,-2,-3}

Dominant Strategies

Let's analyze such games. Some strategies are better

- Dominant strategy: if a_i better than a_i' *regardless* of what other players do, a_i is **dominant**
- I.e.,

$$u_i(a_i, a_{-i}) \geq u_i(a_i', a_{-i}) \forall a_i' \neq a_i \text{ and } \forall a_{-i}$$

All of the other entries of *a* excluding *i*

- Don't always exist!

Another important notion: dominant strategies.

For a player, suppose it discovers that one action a_i is always not worse than the other actions. That is, for any configurations of the other players' actions, a_i leads to no worse outcome than all the other actions.

Dominant Strategies Example

Back to Prisoner's Dilemma

- Examine all the entries: betray dominates
- Check:

Player 2 Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

- Note: normal form helps **locate** dominant/dominated strategies.

Betray is the dominant strategy for Player 1.

When Player 2 stay silent: if Player 1 takes action betray, it gets better outcome
When Player 2 betray: if Player 1 takes action betray, it also gets better outcome

Equilibrium

a* is an equilibrium if all the players do not have an incentive to **unilaterally deviate**

$$u_i(a_i^*, a_{-i}^*) \geq u_i(a_i, a_{-i}^*) \quad \forall a_i \in A_i$$

- All players dominant strategies -> equilibrium
- Converse doesn't hold (don't need dominant strategies to get an equilibrium)

Another important notion is Equilibrium.

In an equilibrium, any player who deviates by itself will suffer.

If all players have dominant strategies, then the configuration of all dominant strategies is an equilibrium, just by the definition of dominant strategies.

Pure and Mixed Strategies

So far, all our strategies are deterministic: "**pure**"

- Take a particular action, no randomness

Can also randomize actions: "**mixed**"

- Assign probabilities x_i to each action

$$x_i(a_i), \text{ where } \sum_{a_i \in A_i} x_i(a_i) = 1, x_i(a_i) \geq 0$$

- Note: have to now consider **expected rewards**

A pure strategy is a special case of mixed strategies: putting probability 1 on one action and 0 on the others.

Nash Equilibrium

Consider the mixed strategy $x^* = (x_1^*, \ldots, x_n^*)$

- This is a **Nash equilibrium** if

$$u_i(x_i^*, x_{-i}^*) \geq u_i(x_i, x_{-i}^*) \quad \forall x_i \in \Delta_{A_i}, \forall i \in \{1, 2, \ldots, n\}$$

Better than doing anything else, **"best response"**

Space of probability distributions

- Intuition: nobody can **increase expected reward** by changing only their own strategy. A type of solution!

In x*, each player has a mixed strategy, a probabilistic distribution over its action space.

Nash equilibrium: all the players do not have an incentive to deviate from its current mixed strategy.

Focus on any player i: If the other players keep their strategies, then player i will suffer from using other mixed strategy.

Properties of Nash Equilibrium

Major result: (Nash 1951)

- Every finite game has at least one Nash equilibrium
  - But not necessarily **pure** (i.e., deterministic strategy)
- Could be more than one!
- Searching for Nash equilibria: computationally **hard**!

Example: rock/paper/scissors has (1/3, 1/3, 1/3) as a mixed strategy NE.

Good: existence
Bad: more than one; hard to compute

Break & Quiz

Q 2.1: Which of the following is true?

(i) Rock/paper/scissors has no dominant pure strategy

(ii) There is no Nash equilibrium for rock/paper/scissors

- A. Neither
- B. (i) but not (ii)
- C. (ii) but not (i)
- D. Both

Break & Quiz

Q 2.1: Which of the following is true?

(i) Rock/paper/scissors has no dominant pure strategy

(ii) There is no Nash equilibrium for rock/paper/scissors

- A. Neither
- **B. (i) but not (ii)**
- C. (ii) but not (i)
- D. Both

Break & Quiz

Q 2.1: Which of the following is true?

(i) Rock/paper/scissors has no dominant pure strategy

(ii) There is no Nash equilibrium for rock/paper/scissors

- A. Neither (i is indeed true: easy to check that there's no deterministic dominant strategy)
- **B. (i) but not (ii)**
- C. (ii) but not (i) (i is true)
- D. Both (ii is false: there exists Nash equilibrium)

Summary

- Review of game theory basics
  - Properties, sequential games
- Speeding up sequential game search
  - Heuristics, pruning, random search
- Simultaneous Games
  - Normal form, strategies, dominance, Nash equilibrium

Acknowledgements: Developed from materials by Yingyu Liang (University of Wisconsin), James Skrentny (University of Wisconsin), inspired by Haifeng Xu (UVA) and Dana Nau (University of Maryland).